Web Services Project

Project 1: REST API (20 points)

You are tasked with building a RESTful API for a social network application.

For that, let’s consider the following Models:

List
Data structure for all responses that return array of data.

```
{
data: Array(Model)
total: number(total items in DB)
page: number(current page)
limit: number(number of items on page)
}
```

User Preview
User as a part of list or other data like post/comment.

```
{
id: string(autogenerated)
title: string("mr", "miss", "dr", "")
firstName: string(length: 2-50)
lastName: string(length: 2-50)
picture: string(url)
}
```

User Full
Full user data returned by id.

```
{
id: string(autogenerated)
title: string("mr", "miss", "dr", "")
firstName: string(length: 2-50)
lastName: string(length: 2-50)
gender: string("male", "female")
email: string(email)
dateOfBirth: string(ISO Date - value: 1/1/1900 - now)
registerDate: string(autogenerated)
phone: string(phone number - any format)
picture: string(url)
location: object(Location)
}
```

Location
Using only as a part of full user data.

```
{
street: string(length: 5-100)
city: string(length: 2-30)
state: string(length: 2-30)
country: string(length: 2-30)
timezone: string(Valid timezone value ex. +7:00, -1:00)
}
```

Post Create
Post data for create request.

```
{
text: string(length: 6-50, preview only)
image: string(url)
likes: number(init value: 0)
tags: array(string)
owner: string(User id)
}
```

Post Preview
Post data as a part of list.

```
{
id: string(autogenerated)
text: string(length: 6-50, preview only)
image: string(url)
likes: number(init value: 0)
tags: array(string)
publishDate: string(autogenerated)
owner: object(User Preview)
}
```

Post
Post data returned by id.

```
{
id: string(autogenerated)
text: string(length: 6-1000)
image: string(url)
likes: number(init value: 0)
link: string(url, length: 6-200)
tags: array(string)
publishDate: string(autogenerated)
owner: object(User Preview)
}
```

Comment Create
Comment data to create new item.

```
{
message: string(length: 2-500)
owner: string(User Id)
post: string(Post Id)
}
```

Comment

```
{
id: string(autogenerated)
message: string(length: 2-500)
owner: object(User Preview)
post: string(Post Id)
publishDate: string(autogenerated)
}
```

Tag
Plain type. Array of strings.

To Do:

1. Design and develop this API using your favorite programming language (framework) to have the following:

   USER:
   Get List
   - Get the list of users sorted by registration date (default sorting criteria).
   - Pagination query params available.

   Get User by id: Get full data of a user by user id
   Create User: Create new user, return created user data.
   Body: User Create (firstName, lastName, email are required)
   Update User: Update user by id, return updated user data
   Body: User data, only fields that should be updated. (it is forbidden to update email)
   Delete User: Delete user by id, return id of deleted user.

   POST:
   Get List
   - Get the list of posts sorted by creation date (default sorting criteria).
   - Pagination query params available.

   Get List By User
   - Get the list of posts for a specific user sorted by creation date.
   - Pagination query params available.

   Get List By Tag
   - Get the list of posts for a specific tag sorted by creation date.
   - Pagination query params available.

   Get Post by id: Get post full data by post id.

Create Post: Create new post, return the created data post.
Body: Post Create (owner and post fields are required)
Update Post: Update post by id, return updated Post data
Body: Post data, (it is forbidden to update the owner field)
Delete Post: Delete post by id, return the id of the deleted post.

COMMENT:
Get List
- Get the list of comments sorted by creation date.
- Pagination query params available.

Get List By Post
- Get the list of comments for a specific post sorted by creation date.
- Pagination query params available.

Get List By User
- Get the list of comments for a specific user sorted by creation date.
- Pagination query params available.

Create Comment: Create a new comment, return created comment data.
Body: Comment Create (owner and post fields are required)
Delete Comment: Delete comment by id, return the id of the deleted post

TAG:
Get List: Get list of tags

a. Choose the right **naming** and **granularity** of different **resources** with respect to the conventional good practices.

b. Choose the right **HTTP verbs** in accordance to each **operation**. (Take into consideration **Safety** and **Idempotence**)

c. Manage the **resources relationships** by respecting the conventional good practices.

d. Manage the following **errors** in accordance to the convenient **server status code**.

> ***PARAMS_NOT_VALID:*** URL params (ex: /user/{id} - {id} is URL param) is not valid. This error returned in both cases: param format is invalid, param is not found.
>
> ***BODY_NOT_VALID:*** Applicable only for not GET requests. Body format is invalid, or even some keys are not valid.
>
> ***RESOURCE_NOT_FOUND:*** Applicable for all requests that have {id} URL param. It means that an item that was requested is not found.
>
> ***PATH_NOT_FOUND:*** Request path is not valid, check controller documentation to validate the URL.

SERVER_ERROR: Something is wrong with server, try again later.

e. Propose various options for:
   i. **Pagination**
   ii. **Sorting**
   iii. **Filtering**
   iv. **Search**

   (You are free to show the options you want)

f. Specify the **version** of this API using two different techniques.

g. Allow **content negotiation** between the client and the API (following a user-driven approach) by proposing at least two different resource representation formats. It's also requested to use coefficients to express the client's preferences.

h. Specify the clients allowed to consume this API using **CORS**.

i. Enable **caching** for GET results. (use a combination of http header parameters)

j. Enable **compression** for GET results. (use at least two different algorithms + allow the client to express its preferences using coefficients)

k. Enable to change the language of the API from English to French using an **i18n** technique. (Do the same for *Date* format)

l. Respect the **Level 3 (HATEOAS)** of **Richardson Maturity Model** when implementing this API.

2. Fully document this API using Swagger UI.

3. Use Swagger CODEGEN to generate the code of this API with another language/framework.

NB: Show the execution of all these features using an HTTP client such as Postman, curl, etc.

Bonus (4 points): You can build your own Web App (or Mobile App) using a JS library/framework to consume this API.

Project 2: GraphQL API (8 points)

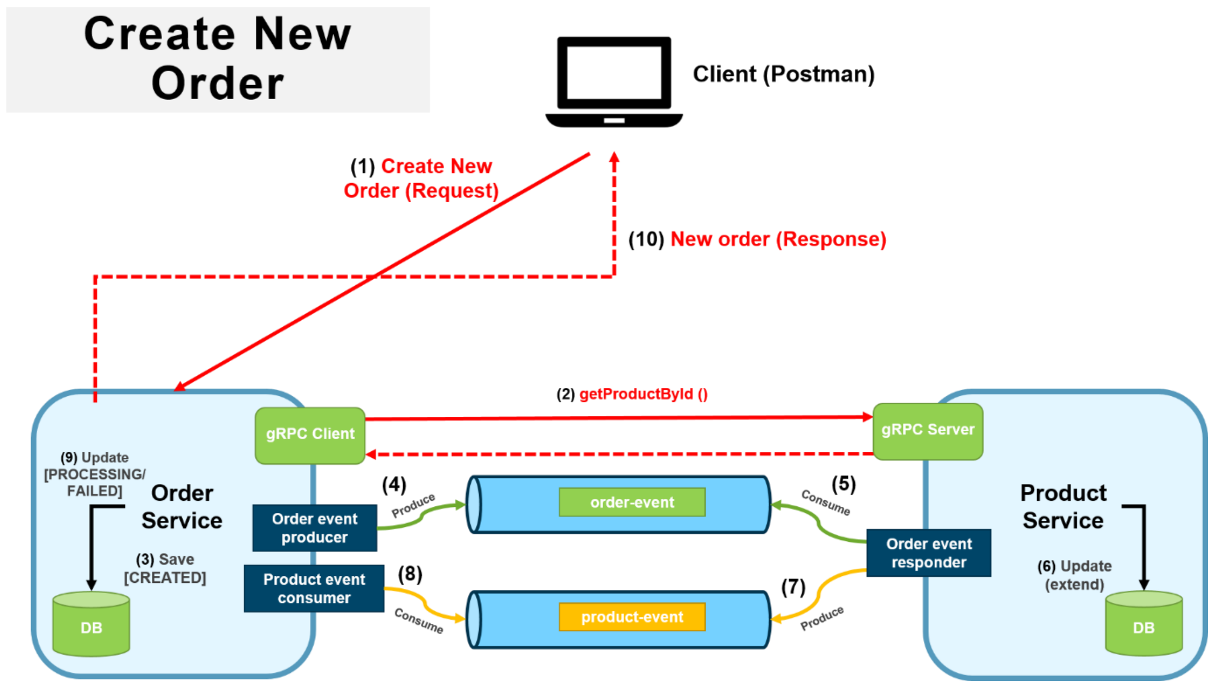
1. After designing and developing the RESTful API (Project 1) you are invited to show and explain a concrete example of **under-fetching** and **over-fetching** of this API.

2. Develop a **GraphQL Server** (using your favorite language / library) that implements the same features of the REST API (while avoiding the problems shown in Question 1)

3. Develop a **GraphQL Client** (using your favorite language / library) that can send queries, mutations and registrations to the GraphQL server developed in Question 2.

Project 3: Asynchronous Messaging + gRPC (12 points)

Let's consider the following Figure:



You are invited to **implement the "Create Order" distributed transaction** which consists of a series of actions.

We assume that an order is made for one product by a *product_id* along with the *quantity*.

The *gRPC* client is calling the method *getProductById()* to get the product data from the *product_service* in order to calculate the order's price.

To continue creating the new order, we need to check first the product availability.

The order_service will receive the new order information from the client, then it will create and save it in the DB with a CREATED order state.

The order_service send then an order event to the product_service in order to check if the product is available or not (the product DB is then updated).

If the product is available, the product service will update the items number in the product DB and then send a product event with the AVAILABLE state to the order_service. If the product is out of stock the product event will gather an OUT_OF_STOCK state.

If the product state is AVAILABLE, the order_service will update the state of the order to PROCESSING, if not it will be changed to FAILED.